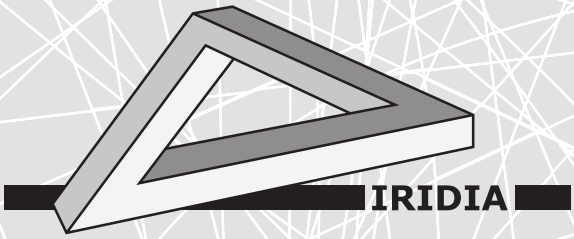

Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

A hash function breaking symmetry in partitioning problems and its application to tabu search for graph coloring

Marco Chiarandini and Franco Mascia

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2010-025

December 2010

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:
IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2010-025

A hash function breaking symmetry in partitioning problems and its application to tabu search for graph coloring*

Marco Chiarandini† Franco Mascia‡

December 2010

Abstract

In partitioning problems we are asked to divide a set of elements into groups such that some properties are satisfied. Local search algorithms solve these problems representing a candidate solution by an array that maps each element to one of the groups. Yet the name of the group is not a relevant information and a simple permutation of the labels of the groups yields an equivalent solution.

In this note we introduce a hash function that takes into account this symmetry of the search space and can be used to prevent revisiting solutions that differ only for a permutation of group labels. We use the function to store and recognize solutions in a tabu search algorithm for graph coloring and we compare it experimentally with a more common version of this algorithm that simply prevents the reintroduction of a single attribute.

1 Introduction

Several combinatorial optimization problems are of partitioning type: given a set of elements find a partition of the set into a number of subsets with each of the subsets having to meet the same requirements. These problems exhibit an intrinsic symmetry, since the subsets are indistinguishable and a permutation of them does not alter the solution. The theory on permutation groups formalizes this concept [3]. Examples of such problems are graph coloring, scheduling of jobs on parallel identical machines and experimental design theory.

In complete solution techniques such as mixed integer programming and constraint programming, partitioning problems are conveniently modelled as assignment problems and solutions to them defined by a mapping from the elements to the groups, i.e. from variables to values. In practice this is realized by binary arrays of pairs variable–value or by arrays of integer numbers in which each element represents the value assigned to the corresponding variable. The symmetry is maintained

*This work was carried out while both authors where visiting researchers at IRIDIA, Université Libre de Bruxelles

†✉ IMADA, University of Southern Denmark, Campusvej 55, DK-5230 Odense, Denmark.
✉ marco@imada.sdu.dk

‡✉ DISI, Università di Trento, via Sommarive 14, I-38123 Trento, Italy.
✉ mascia@disi.unitn.it

in this representation in that a permutation of the values preserves the solution quality and the set of constraints.

Models that hold solution symmetries lead to a search space that is larger than necessary thus possibly affecting negatively the solution time. In mixed integer programming and in constraint programming where a search tree has to be examined completely this is often bad and a considerable body of literature has proposed remedies to this issue. In mixed integer programming the main approaches to deal with symmetries are perturbation of input data, variable fixing, (isomorphism) pruning of the search tree, symmetry breaking inequalities and orbital branching [14]. In constraint programming the approaches are problem reformulation, constraint addition, and dynamic symmetry breaking during the search process via, again, constraint addition and solution dominance detection [19]. In dynamic symmetry breaking detection procedures need to be devised.

The effect of symmetries on incomplete solutions techniques is less certain. In the particular case of local search techniques that represent solutions as complete candidate assignments of variables to values, it is conjectured, and experimentally supported, that symmetry breaking constraints can create additional local optima thus reducing the basin of attraction of global optima, and that they can make the search space disconnected [18]. On the other side, using search diversification mechanisms such as those introduced by the metaheuristics, symmetry may become a problem. For example, with genetic algorithms for partitioning problems computing the exact distance between solutions that accounts for symmetries, thus detecting and avoiding early convergence of the search, has received considerable attention. As evidence of this fact an algorithm for computing this distance efficiently has been rediscovered several times in the recent years [8],[10],[9],[4, p. 131],[20],[17].

One of the most successful metaheuristics for local search in partitioning problems is tabu search. The main idea is to continue the search beyond local optima while allowing to accept neighboring solutions that are not better than the current one but forbidding to revisit already seen solutions. The implementation of the prohibition criterion is the crucial element of this metaheuristic. In a one-exchange local search where the neighborhood of a solution is defined by all solutions that differ from the current one by only one single variable–value assignment, the common procedure is to forbid the single variable–value reassignment for a number of later iterations. This prohibition, although fast to check, does not prevent from revisiting equivalent solutions because of symmetries. Most importantly, it excludes many more solutions than just the previously visited ones and hence each single prohibition must remain active for a limited number of iterations. This number of iterations, also called tabu tenure, is an instance dependent parameter that requires experimental tests for its tuning.

In this note, we aim at designing a data structure for local search to store and check already visited solutions and all their symmetric equivalent, using linear space and performing *find* and *insert* operations in a constant expected execution time. We achieve this by devising an universal hash function that can be incrementally updated in constant time and that returns the same value for all equivalent partitioning of the elements. However, in long runs of the local search, storing hashed solutions can deteriorate to linear search.

2 An universal hash function

Let $E = \{1, 2 \dots |E|\}$ be the set of elements and $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$ the set of subsets of elements. We call candidate solution, and represent it by a mapping $\varphi : E \mapsto \mathcal{C}$, any partitioning of the elements in the subsets. A candidate solution needs not necessarily to satisfy all requirements on the subsets but any solution to the partitioning problem is a candidate solution. For example, a candidate solution in the vertex k-coloring problem [12] is any assignment of exactly one color to all the vertices of the graph including colorings that assign the same color to adjacent vertices. In practice, φ is realized by a simple array of integer numbers, each number indicating the subset to which an element is assigned.

We use an associative array S to store the set of solutions already visited representing each solution φ by a key $key(\varphi) \in \mathcal{K}$. Distinct candidate solutions have distinct keys. The number of potential candidate solutions $|\mathcal{K}| = N$ is clearly huge, namely $k^{|E|}$. We aim at a storage S of smaller size m, sufficient to contain the number of visited solutions n.

We seek a hash function to map the set $\mathcal{K}$ to a smaller range of integers $0, \dots, m-1$ and store the solutions in a *hash table* t consisting of an array with index set $0, \dots, m-1$. The hash function associates to each solution φ a *hash value* $h(key(\varphi))$ and stores the key in the table entry $t[h(key(\varphi))]$. Since $N > m$ there will be elements that map to the same entry. This leads to the possibility of collisions that we resolve by *chaining* [6]: when two distinct solutions with different hash function h are mapped to the same position j in the array t, we append them both in a list pointed by $t[j]$. If there are no elements $t[j]$ contains the null pointer. Clearly, a good hash function is one that maps each key in any of the m slots with equal probability, independently of where any other key has hashed to.

In the following, we present an universal hash function that is easy to evaluate and does not require more than linear space to store. In addition, it has the appealing property that it maps to the same hash value candidate solutions that differ only by a permutation of group identifiers. We choose m to be a prime (for the nice arithmetical properties of the operation modulo a prime see for example [6, Chp. 31]). For a candidate solution φ we define $\boldsymbol{a} = (a_1, \dots, a_{|E|})$ to be a vector of integers

$$a_i = (\lfloor \phi^{-1} w \rfloor \cdot i) \bmod w \qquad \forall i = 1, \dots, |E|,$$

where ϕ^{-1} is the inverse of the golden ratio and w another prime number, and compute for each element the value

$$x_i = (a_i \cdot i) \bmod m \qquad \forall i = 1, \dots, |E|,$$

We subdivide the $key(\varphi)$ into a k-tuple of integers $\boldsymbol{y} = (y_1, \dots, y_k)$ defined by the sum of the values of the elements in the subsets, i.e.

$$y_j = \sum_{i \in C_j} x_i \qquad \forall j = 1, \dots, k.$$

Further, for each subset we define $\boldsymbol{b} = (b_1, \dots, b_k)$

$$b_j = (\lfloor \phi^{-1} w \rfloor \cdot y_j) \bmod w \qquad \forall j = 1, \dots, k$$

and finally compute the hash value of the candidate solution

$$h(key(\varphi)) = \left(\sum_{j=0}^{k} b_j \cdot y_j \right) \bmod m = \boldsymbol{b} \cdot \boldsymbol{x} \bmod m.$$

The hash function $h(key(\varphi))$ is universal because defined by the scalar product between a random vector and a tuple representation of a key modulo a prime [15, Th. 4.4]. Moreover, by the definition of b_j and the commutativity property of addition, its value does not change if the subsets are just permuted.

Computing $h(key(\varphi))$ takes $O(|E|)$. However, when a solution changes only in the subset to which a single element is assigned, as in the case of the one-exchange neighborhood for local search, the new hash value can be computed in constant time. Indeed, if a new solution φ' is obtained from φ by moving one element e from subset p to subset q we have

$$\begin{aligned} h(key(\varphi')) = h(key(\varphi)) & \\ - ((\lfloor \phi^{-1} w \rfloor \cdot y_p) \bmod w) \cdot y_p & \\ + ((\lfloor \phi^{-1} w \rfloor \cdot y'_p) \bmod w) \cdot y'_p & \\ - ((\lfloor \phi^{-1} w \rfloor \cdot y_q) \bmod w) \cdot y_q & \\ + ((\lfloor \phi^{-1} w \rfloor \cdot y'_q) \bmod w) \cdot y'_q & \end{aligned}$$

where $y'_q = y_q + x_e$ and $y'_p = y_p - x_e$. The constant time derives from the fact that $\boldsymbol{x} = (x_1, \ldots, x_{|E|})$ can be stored and does not need to be recomputed. Indeed, $\boldsymbol{x}$ does not change during the search, always mapping elements to the same values, and it can be stored in a space that is linear in the number of elements. Note that storing $\boldsymbol{b}$ would require much more space, because each b_j is a function of the integers that identify the specific elements contained in the subset C_j and hence we would need to keep m different values.

In general, then, the storage S can have size d, with $d < m$, and we need a further map of $h(key(\varphi))$ to $0, \ldots, d-1$, which can be done by $h(key(\varphi)) \bmod d$. In our experiments we will fix d to be the same constant large prime as m, thus saving the last modulo operation. To guarantee constant expected execution time for *find* and *insert* operations in the hash table we need $m = \Theta(n)$ (see, for example, [15, Th. 4.3]). However n, the number of visited solutions, is continuously increasing during the local search and not known to be bounded by a polynomial [16, Chp. 6]. Hence, deciding the value for m, we will have to accept that the search deteriorate to expected linear time when the number of visited solutions grows much larger than m.

3 Engineering tabu search for graph coloring

We used the hashing described in the previous section in two enhanced versions of the well known tabu search algorithm for graph coloring, dubbed tabucol [11]. We call these two versions tabuhash and tabu_reactive.

More in detail, we address the optimization version of the vertex coloring problem, in which the task is finding a proper coloring of the vertices that uses the least

number of colors. The common approach with local search techniques is to solve a series of decision problems with decreasing number of colors. Each decision problem is in turn solved by minimizing to zero the number of edges in conflict.

Let $G(V, E)$ be a general graph and k the number of colors. The common elements to all three tabu search algorithms are as follows. A starting solution is constructed by means of the RLF heuristic [13]. When a proper coloring is found the number of colors k is decreased and vertices reassigned at random to the other color classes. For each fixed k a candidate solution is evaluated by the number of edges that create conflicts in the coloring, i.e., $f(\varphi) = \sum_{i=1}^{k} |E(C_i)|$, where $E(C_i)$ is the edge set of the subgraph of G induced by C_i. The one-exchange neighborhood defines as neighbors two solutions that can be obtained from each other by moving one vertex v from a color class C_j, $j \in \{1, \ldots, k\}$ into a different color class C_l, $l \neq j$. This possibility is restricted to only vertices that are involved in some conflict violation. The evaluation of a neighbor after a one-exchange is computed by $f(\varphi') = f(\varphi) - |A_{C_i}(v)| + |A_{C_j}(v)|$, where $A_{C_i}(v)$ is the set of vertices adjacent to v in the subgraph induced by C_i. An additional $|V| \times k$ matrix is used to store the values $|A_{C_i}(v)|$ and updated in $\mathcal{O}(|V|)$ after each iteration. Thus each neighbor is evaluated in constant time. The three tabu search versions differ in the way they select the next solution to visit during their search.

Tabucol It chooses a best non-tabu neighboring candidate solution. If the color class of vertex v changes from C_i to C_j, it is forbidden to assign color i to vertex v in the next tt steps. This prohibition is broken only when such a move would lead to an improvement over the best candidate solution encountered so far (aspiration criterion). The tabu tenure, tt, for a specific vertex–color pair (v, i) is set proportional to the size of the neighborhood as $tt = random(10) + \alpha \cdot 2 \cdot |E_V^c|$ where $random(10)$ is an integer uniformly chosen from $\{0, \ldots, 10\}$. Hence, the tabu tenure varies dynamically at run-time depending on the evaluation function value (similar dynamic tabu tenures are used by [7] while they are not present in the original implementation of [11]). To avoid forbidding too many solutions, tt must be at least smaller than the neighborhood size. Hence, $\alpha \ll k$. We implement the tabu list by a $k \times |V|$ matrix where we write the iteration number until which the move is tabu. If more than one move produces the same effect on the evaluation function, one of these is selected randomly. Extensive preliminary experiments and literature results indicate that $\alpha = 0.6$ is a robust enough setting.

Tabuhash We use the hash table to determine the tabu status of a solution considered to be visited. In the examination of the neighborhood, whenever a neighbor is considered whose quality is at least as good as the best neighbor seen so far, its presence in the hash table is tested and kept as candidate for selection only if it is not present in the table.

Note that both checking the vertex-color attribute in the matrix in tabucol and computing the hash value and accessing the hash table in tabuhash can be done in constant time (although involving a different number of operations). However when collisions arise the retrieval of the hash value in the table may require linear time.

Tabu_reactive Reactive mechanisms (see e.g. [1]) that change the value of α during the search are an alternative approach to tuning off-line the value of the parameter α in tabucol. An attempt in this direction has been proposed in [2]; however the scheme is based solely on the values of the evaluation function and it is therefore rather simplistic. The experiments reported in that work do not make clear whether the mechanism is effective or not.

In our reactive scheme, the prohibition criterion is based on the single attribute as in tabucol and the tabu length is a constant. Initially, the tabu length is set to 1. If, after a move has been selected and performed, the new solution is detected to be in the hash table and hence already visited, then the tabu tenure tt is increased to $\lfloor tt \cdot 1.01 \rfloor$. The increase is not performed when the value makes tt larger than the size the neighborhood $|V^c|k$. When no solution examined during the search is detected in the hash table for more than $|V|$ iterations, the tabu tenure is decreased to $\lceil tt \cdot 0.99 \rceil$, if this value does not fall below 2.

Note that this procedure does not completely remove parameters but it is less sensible to the meta-parameters it depends on. In other terms, we expect to improve the performances with respect to tabucol without need for tuning. Note also that while in tabuhash we need to compute the hash value and access the hash table at each neighbor that is evaluated, in tabu_reactive we need to do this operation only once at each iteration, namely for the neighboring solution that has been chosen.

4 Computational experiments

We implemented the three tabu search algorithms sharing common data structures in a C++ framework. We realized the hash table t as a vector of lists using the data structures from the C++ Standard Template Library. Further, we set $\phi^{-1} = 0.618033$, $w = 1200007$, $d = m = 1200061$ and used 64 bits integers to avoid overflows in the operations to compute the hash values. This entails that the initial hash table without chaining requires about 10 MB of memory independently from instance size.

We tested the three tabu search variants on uniform random graphs. We generated 5 graphs for each combination of size $\{500, 1000, 2000\}$ and edge density $\{0.1, 0.5, 0.9\}$ and run once the three algorithms on each instance with relative long run times, since we would expect that the variants introduced improve tabucol when the last is not anymore effective. As time limit we then used the median time tabucol took to perform $10^5|V|$ total iterations. The median is computed for each of the 9 instance classes thus yielding a different time limit for each class.

We run the experiments on machines with 2 INTEL Xeon E5410 (quad core 2.33GHz, 2x 6MB L2 cache) that share 8GB RAM. Source code was compiled on this architecture with the GNU C++ compiler version 4.4.0 and flags `-O3 -march=native -ffast-math`. As an idea, on the largest and most dense graphs the time limit we used was larger than 6 hours.

In Figure 1, we show the mean rank of the results found by the three algorithms distinguished per instance class. Around the mean we plot the confidence intervals obtained by a post-hoc, all-pairwise analysis carried out by way of the Friedman test [5]. The ranks of the results achieved by two algorithms cannot be claimed different with statistical significance if their confidence intervals overlap. The figure

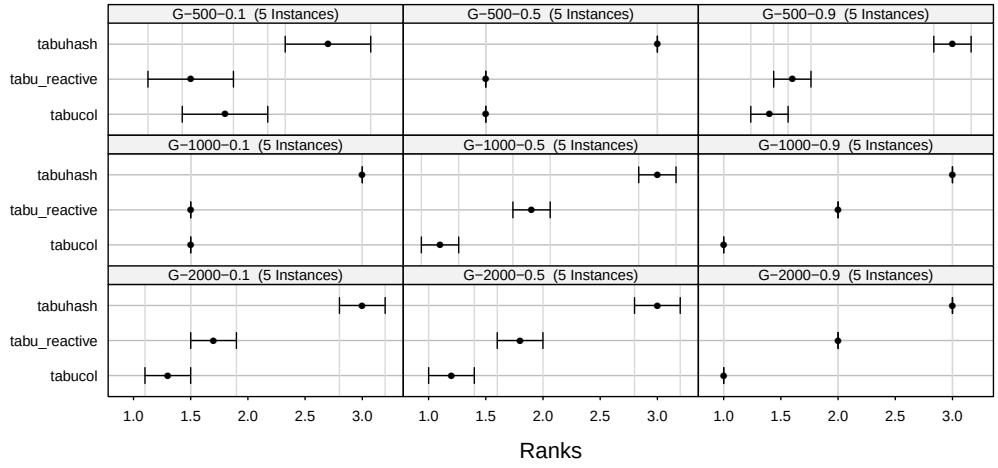

Figure 1: Paired plot analysis. For each instance class we visualize by means of confidence intervals the results of the all-pairwise comparisons conducted by means of the Friedman test. The dots around which the confidence intervals are attached correspond to the average rank obtained by the algorithms. Ranks are computed "within" the instances and aggregated "between" instances. Since we collect one single run per algorithm on each instance, ranks go from 1 to the number of present algorithms. The difference between two algorithms is statistically significant if their corresponding intervals do not overlap.

show that tabuhash and tabu_reactive are never the best. An aggregated analysis (not shown) that would not distinguish among instance classes indicates as clear winner tabucol.

In Figure 2 we give evidence of a possible explanation for this result. The total number of iterations, corresponding to the total number of solutions visited during the search, is much higher for tabucol than for the two other algorithms. In fact tabuhash achieves even less iterations than tabu_reactive. There is therefore a clear correlation between these numbers and the ranking of the three algorithms.

Further, we compared the size of the hash table m with the number of visited solutions n to gain insight on the actual time of hashing operations. In Figure 3 we show the number of iterations performed at the last number of colors that tabuhash tried to solve without success in each instance of the the k-coloring problem before terminating. The number of iterations corresponds to the number of solution keys to store in the hash table and the figure indicates that this number is about 10 or 100 times the initial size m of the table. Hence, chaining occurs and the average number of operations for *find* and *insert* is between 10 and 100 . This may become onerous if repeated at each iteration and explains why tabuhash and tabu_reactive achieve less iterations overall.

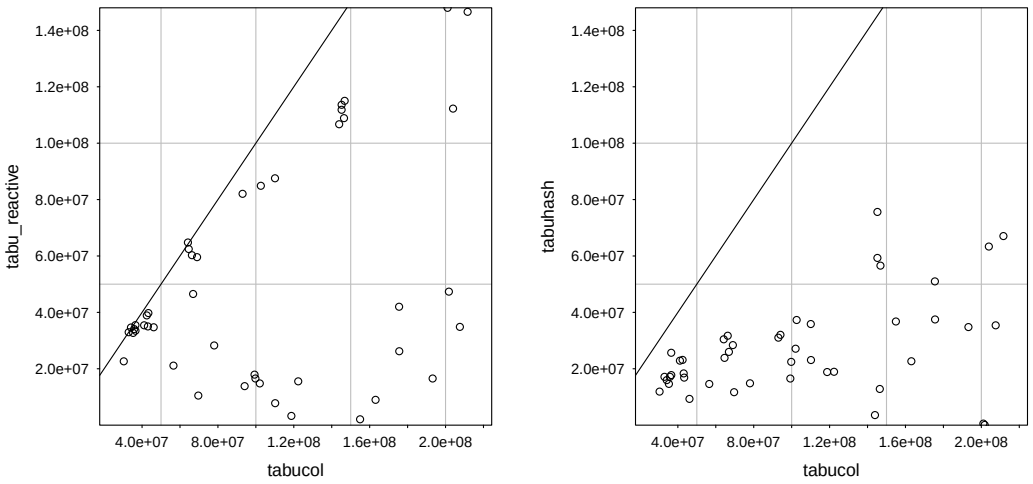

Figure 2: In the left plot the number of total iterations of tabucol (vertical axis) is plotted against the number of iterations of tabuhash and tabu_reactive (horizontal axis of the left and right plot, respectively) observed on each instance in a fixed amount of time.

5 Discussion

We introduced a hash function to be used for storing efficiently solutions visited during local search in partitioning problems. The function maps on the same values solutions that are symmetric and hence equivalent, thus providing with a symmetry breaking mechanism. In addition, we showed that if the neighborhood is defined by the one-exchange operator, then the hash value of a neighboring solution can be computed in constant time from the value of the current solution.

In spite of these appealing properties the use of a hash table within tabu search for graph coloring is not helpful. If each neighbor is tested for presence in the hash table, at parity of total computation time, we obtain a clear worsening of performance. A more parsimonious use of the hash table accessed only once at each iteration, as in the case of the reactive mechanism tested above, is also not successful. The explanation is the deterioration of the hashing operations to linear time when the number of iterations and, consequently, of visited solutions becomes large and for an hash table of size m, $m = \Theta(n)$ does not hold anymore. Clearly, at parity of time, higher time spent in hashing operations implies that less solutions are visited and this may explain why tabucol that performs more iterations achieves better results. A further possible attempt could be to increase m or reinitialize the hash table when a number of iterations multiple of m, say 2, 3, has been achieved. This would ensure again constant expected time for hashing operations while possibly not affecting seriously the quality of the final solutions. It would be interesting, however, also studying the amount of repeated configurations effectively visited during the search by tabucol. We conjecture that this number remains low and not significant to affect negatively the search. In spite of the preliminary results here presented, the hashing function introduced and the possibility to compute its value in constant time in local search algorithms are, at least in line of principle, appealing features

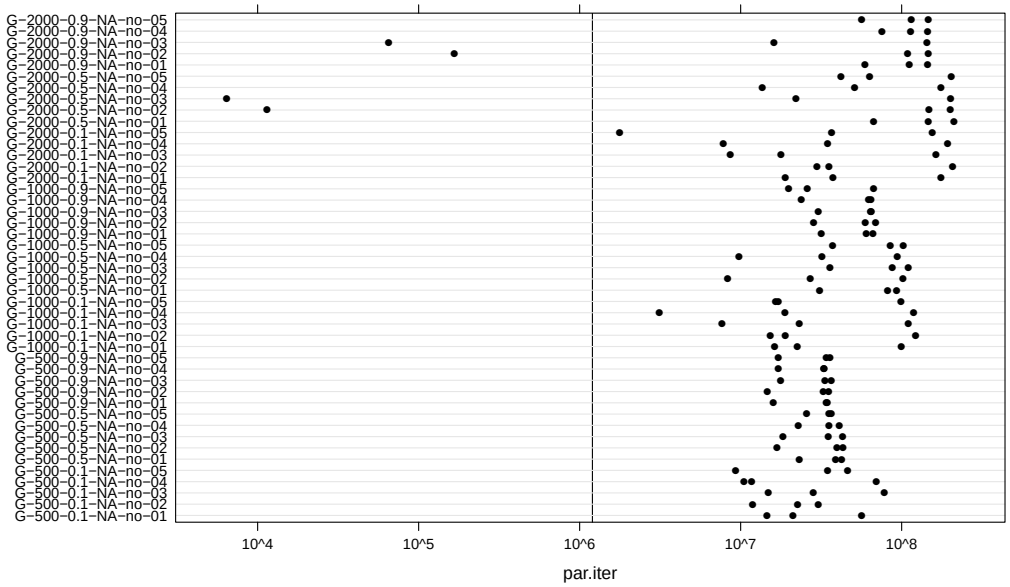

Figure 3: For each instance on the y-axis we plot the number of iterations performed by tabuhash when solving the last of the k-coloring problems. This number corresponds to the number of solutions to store n. The vertical line represents the initial size m of the hash table.

and their possible contribution may perhaps be worth testing on other problems.

References

[1] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*. Operations research/Computer Science Interfaces. Springer Verlag, 2008.

[2] I. Blöchliger and N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35(3):960–975, 2008.

[3] J. P. Cameron. *Permutation groups*. Cambridge University Press, 1999.

[4] M. Chiarandini. *Stochastic Local Search Methods for Highly Constrained Combinatorial Optimisation Problems*. PhD thesis, Computer Science Department, Darmstadt University of Technology, Darmstadt, Germany, August 2005.

[5] W. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, New York, NY, USA, third edition, 1999.

[6] T. Cormen, C. Leiserson, R. Rivest, and Stein. *Introduction to algorithms*. MIT press, Cambridge, Massachusetts, USA, second edition, 2001.

[7] R. Dorne and J. Hao. A new genetic local search algorithm for graph coloring. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel*

Problem Solving from Nature - PPSN V, 5th International Conference, volume 1498 of *Lecture Notes in Computer Science*, pages 745–754. Springer Verlag, Berlin, Germany, 1998.

[8] P. Galinier and J. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

[9] C. A. Glass and A. Prügel-Bennett. A polynomially searchable exponential neighbourhood for graph colouring. *Journal of the Operational Research Society*, 56(3):324–330, 2005.

[10] D. Gusfield. Partition-distance: A problem and class of perfect graphs arising in clustering. *Information Processing Letters*, 82(3):159–164, 2002.

[11] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[12] T. R. Jensen and B. Toft. *Graph coloring problems*. Wiley Interscience, New York, USA, 1995.

[13] F. T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–506, 1979.

[14] F. Margot. Symmetry in integer linear programming. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 647–686. Springer Berlin Heidelberg, 2010.

[15] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[16] W. Michiels, E. Aarts, and J. Korst. *Theoretical Aspects of Local Search*. Monographs in Theoretical Computer Science. Springer, 2007.

[17] D. Porumbel, J.-K. Hao, and P. Kuntz. An efficient algorithm for computing the distance between close partitions. *Discrete Applied Mathematics*, 159(1):53–59, 2011.

[18] S. D. Prestwich and A. Roli. Symmetry breaking and local search spaces. In R. Barták and M. Milano, editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2005.

[19] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[20] E.-G. Talbi and B. Weinberg. Breaking the search space symmetry in partitioning problems: An application to the graph coloring problem. *Theoretical Computer Science*, 378(1):78 – 86, 2007.